AD676 017

TO APPEAR IN THE JOURNAL
OF THE A.C.M.

A FORMAL DEDUCTIVE PROBLEM-SOLVING SYSTEM

J. R. Quinlan and E. B. Hunt

This research was partially supported by the National Science Foundation. Grant No. NSF B7-1438R, to the University of Washington, Earl Hunt, Principal Investigator, and partially by the Air Force Office of Scientific Research, Office of Aerospace Research, United States Air Force, under AFOSR Grant No. AF-AFOSR-1311-67. Distribution of this document is unlimited.

Computer Science Group
University of Washington -- Seattle
Technical Report No. 68-1-01
February 9, 1968

A FORMAL DEDUCTIVE PROBLEM-SOLVING SYSTEM[1]

J. R. Quinlan and E. B. Hunt
The University of Washington

1. Introduction

1.1 Motivation Computer programs which solve problems have excited considerable interest. Two lines of research have been followed. In one, deductive theorems are translated into statements in a first-order theory, and a first-order proof sought. The work of J. A. Robinson (10) exemplifies this approach. An alternative procedure is to represent a deductive problem as a problem of finding permissible rewriting rules which permit one to go from a string of symbols describing the initial state to a string of symbols describing the goal state. The "General Problem solver" (GPS) of Newell, Shaw and Simon, (7,8), although not originally presented in these terms, is an example of this sort of program.

The internal operation of the GPS-like theorem provers resembles the syntactic phase of a compiler, in that production rules are applied to change sentences from one form to another.

In this paper, we shall present a formal description of a class of problem-solving programs of the compiler-like variety. It will specify a class of recursive programs quite similar to "top-down" compiler-compilers. We shall then show that there is a class of problems which such programs cannot solve. Next we describe a modified structure which generates more powerful programs. These are non-recursive programs which destroy their similarity to a compiler-compiler.

Our work was originally motivated by descriptions of the GPS, and we acknowledge our intellectual debt to Newell and his co-workers. We have proceeded in a somewhat different direction, however, and our present system should not be regarded as a formalization of GPS. We shall return to the point after presenting our system. Programs generated within our system have been written in Algol and FORTRAN IV, and have been used to solve reasonably difficult problems. Some examples will be presented.

1.2 General Description: States are represented by strings of symbols. Strings obeying certain rules for permissible sequences of symbols are called structures, or well formed expressions. A state is always represented by a structure. Structures may contain substructures. The symbols of a string are drawn from a infinite, countable set of symbols called the alphabet.

A transformation writes one well formed expression into another. A problem is solved when a sequence of transformations is found that writes the starting state into the goal state.

2. Definitions

2.1 Alphabet: The alphabet, $\underline{A}$, is an infinite, countable set of symbols formed by the union of a finite set $\underline{T}$ of terminal symbols and an infinite, countable set $\underline{V}$ of non-terminal symbols.

2.2 Terminal Symbols: The set of terminal symbols $\underset{\sim}{T}$, is the union of $\underset{\sim}{m}$ finite sets $\underline{T}_{\underline{i}}$ of terminal symbols of degree $\underline{i}$. That is,

$$\underline{T} = \cup\ \underline{T}_{\underline{i}}\ ,\ \underline{i} = 0\ ,\ \ldots\ \underline{m} - 1$$

where, if $\emptyset$ is the empty set,

$$\underline{T}_{\underline{i}}\ \cap\ \underline{T}_{\underline{j}} = \emptyset \text{ if } \underline{i} \neq \underline{j}\ .$$

Example: In elementary algebra, $\underline{T}$ would consist of the set $\underline{T}_0$ of bound variable symbols $\{\underline{a}, \underline{b}, \underline{c}, \text{etc.}\}$, the set $\underline{T}_1$ containing unary minus, and the set $\underline{T}_2$ of binary connectives $\{+\ ,\ -\ ,\ \times\ ,\ /\ ,\ \uparrow\}$.

2.3 Non-Terminal Symbols: The set $\underline{V}$ consists of the infinite countable set of symbols $\{V_{\underline{i}}, \underline{i} > 0\}$, which denote free variables.

2.4 Terminal Classes: Let $\underline{C} = \{C_{\underline{i}}, \underline{i} = 0, \ldots \underline{n} - 1\}$ be a set of sets of terminal symbols, such that

(i) all the elements in $C_{\underline{i}}$ are of the same degree for all values of $\underline{i}$, and

(ii) every terminal symbol is in at least one $C_{\underline{i}}$.

2.5 Structures (well formed expressions): Well formed expressions are defined by the following rules:

2.5.1: A variable symbol is a well formed expression.

2.5.2: A terminal symbol of degree $\underline{n}$ followed by $\underline{n}$ well formed expressions is a well formed expression.

This is simply the prefix or "Polish" notation. It is obviously isomorphic to a tree graph.

3. Relations between structures:

3.1 Terms describing strings and substrings: Let $\underline{s}$ be a string of symbols $s_0, s_1, s_2, \ldots, s_i, \ldots, s_{n-1}$.

3.1.1: The length of $\underline{s}$ $(L(\underline{s}))$ is $\underline{n}$.

3.1.2: The symbol s_i is the symbol in position $\underline{i}$ of $\underline{s}$, starting with s_0.

3.1.3: If $\underline{s}$ is a structure, $\underline{s(i)}$ is the well formed expression (substructure) beginning with symbol s_i

3.2 Concatenation: Let $\underline{s}^a$ and $\underline{s}^b$ be strings. $\underline{s}$ is a concatenation of $\underline{s}^a$ and $\underline{s}^b$ if

(1) $\underline{s} = s^a_0, s^a_1, \ldots s^a_{n-1}, s^b_0, s^b_1, \ldots s^b_{m-1}$

where $L(\underline{s}^a) = \underline{n}$ and $L(\underline{s}^b) = \underline{m}$. This is written

(2) $\underline{s} = \underline{s}^a \underline{s}^b$.

3.3 Direct components. For any structure $s(i)$, let $N(s(i)) = s(i + L(s(i)))$, i.e., the structure immediately following $s(i)$ in s. $N(s(i))$ is the following structure of $s(i)$, similarly $N^2(s(i))$ is the following structure of the following structure of $s(i)$. Let s_{j-1} be a symbol of degree n. The direct components of s_{j-1} are $s(j)$, $N(s(j)), \ldots, N^{n-1}(s(j))$.

3.4 Equivalence. Two strings s and t are equivalent if $L(s) = L(t)$, and for every i, $0 \leq i < L(s)$, either

(a) $s_i = t_i = V_j$ for some j, or

(b) $s_i, t_i \in S_k$ for some k.

3.5 Substitution: The value of the function $B(V_i, u, s)$ is defined as the string which results from replacing each occurrence of the symbol V_i in the string s with the well formed expression u. The rule of substitution states that a structure s may be replaced by the structure $B(V_i, u, s)$ for any V_i and any well formed expression u.

3.6 Specification: A string s is a specification of a string t (written $s \leq t$) if there is a set of pairs $\{(V_{i_j}, u_j), j = 1, n\}$ such that the strings s and

$B(V_{i_n}, u_n, B(V_{i_{n-1}}, u_{n-1}, B(\ldots, B(V_{i_1}, u_1, t) \ldots)))$ are equivalent.

Comment: In the degenerate case, equivalence and specification apply to symbols.

3.7 Correspondence Given two strings s and g, the correspondence relation C between their symbols is defined as follows:

(a) $s_0 \, C \, g_0$ (s_0 corresponds to g_0)

(b) $s_j \subset s_k$ if there exist j', k' such that

(i) $s_{j'} \subset s_{k'}$,

(ii) $s_{j'}\ S\ s_{k'}$, and

(iii) $s(j)$, $s(k)$ are the ith direct components of $s_{j'}$, $s_{k'}$, respectively.

4. <u>Rewriting rules and operators:</u>

4.1 <u>Metalinguistic symbol ":="</u> : The metalinguistic symbol ":=" is defined to mean "may be rewritten as". The statement $s := s''$ is interpreted as "structure s may be rewritten as structure s^*".

4.2 <u>Rewriting rules</u>: Let l_i and r_i be structures. The rule $l_i := r_i$ is rewriting rule R_i .

4.3 <u>Application of rewriting rules</u>: Let s be a structure, and R_i a rewriting rule. The application of R_i to s , written $R_i(s)$, is defined by

(a) if s is not a specification of l_i , $R_i(s)$ is undefined.

(b) if s is a specification of l_i , let Z be the sequence of substitutions which transforms l_i into a string equivalent to s i.e., s is equivalent to $Z(l_i)$. Then $R_i(s) = Z(r_i)$.

<u>Example</u>: Suppose R_1 is $+\ V_1\ V_2 := +\ V_2\ V_1$, and an attempt is made to form $R_1(\ +\ a\ +\ b\ c\ )$. The required substitution sequence Z can be defined by

$$Z(l_1) \equiv B(V_2,\ +\ b\ c\ ,\ B(V_1,\ a,\ l_1)) = +\ a\ +\ b\ c\ ,$$ so $R_1(\ +\ a\ +\ b\ c)$

is defined as the structure

$$Z(r_1) \equiv B(V_2,\ +\ b\ c\ ,\ B(V_1,\ a,\ r_1)) = +\ +\ b\ c\ a\ .$$

4.4 <u>Operators</u>: An operator, O_{ij} is applied to structure s when rewriting rule R_i is applied to substructure $s(j)$ of s . Let $k = L(s(j))$,

$\underline{n} = L(\underline{s})$. Then

(a) $0_{\underline{ij}}(\underline{s})$ is undefined if $\underline{s(j)}$ is not a specification of $\underline{\mathit{l}_i}$.

(b) $0_{\underline{ij}}(\underline{s}) = s_0 s_1 \ldots s_{j-1}\ \underline{R_i}(\underline{s(j)})\ s_{j+k}\ s_{j+k+1} \ldots s_{n-1}$ otherwise.

4.5 Analysis of the effects of rewriting rules: In order to apply rule $\underline{R_i}$ to some structure $\underline{s}$, one must find a substitution sequence $\underline{Z}$ which satisfies 4.3(b). The result of applying the rewriting rule will be the structure $\underline{s}^* = \underline{Z}(\underline{r_i})$. For every symbol in $\underline{r_i}$, there will be a symbol or a structure in $\underline{s}^*$ which is a specification of it. Let $(\underline{r_i})_j$ be the $\underline{j}$th symbol in $\underline{r_i}$. If $(\underline{r_i})_j$ is a terminal symbol, then this symbol will appear in $\underline{s}^*$, and so one of the symbols of $\underline{s}^*$ is therefore fixed. If $(r_i)_j$ is a non-terminal symbol, then the corresponding structure of $\underline{s}^*$ is defined only in terms of the symbols in $\underline{s}$.

In analyzing rewriting rules, two tables can be constructed. One lists the symbols of $\underline{s}^*$ whose identity is fixed regardless of the symbols in $\underline{s}$, while the second table lists the structures of $\underline{s}^*$ which are defined in terms of $\underline{s}$. (In effect, this is what happens. Actually, the structures of $\underline{r_i}$ which are defined in terms of structures of $\underline{\mathit{l}_i}$ are listed.)

Example: Consider the distributive law of algebra.

$$\underline{+}\ \underline{x}\ \underline{V_1}\ \underline{V_2}\ \underline{x}\ \underline{V_3}\ \underline{V_2} := \underline{x}\ \underline{+}\ \underline{V_1}\ \underline{V_3}\ \underline{V_2}\ .$$

Whenever this rule is applied, the first symbol of the resulting structure is always "$\underline{x}$" and the second always "$\underline{+}$" . The structure beginning at the third symbol will be whatever structure began at the symbol corresponding to $\underline{V_1}$ in the left hand string, and similarly for the structures replacing $\underline{V_2}$ and $\underline{V_3}$. The results of applying a particular operator $\underline{0_{ij}}$ to string $\underline{s}$ can be determined by examining $\underline{s(j)}$ to find what the specifications of $\underline{V_1}$, $\underline{V_2}$, and $\underline{V_3}$ are in this case.

__Comment__: The tables defined by analyzing rewriting rules play the same role in our formalism as do operator-difference tables in the GPS (8).

5. __Problems defined__:

5.1 __Difference sets__: Let $\underline{s}$ and $\underline{g}$ be two strings. The difference set of $\underline{s}$ and $\underline{g}$, $E(\underline{s},\underline{g})$, is defined by

$$E(\underline{s},\underline{g}) = E^*(\underline{s},\underline{g}) \bigcup_j E^j(\underline{s},\underline{g}), \text{ where}$$

(a) A pair (g_j, k) is a member of $E^*(\underline{s},\underline{g})$ if

(1) $s_k \subset g_j$

(2) g_j is __not__ a variable symbol

(3) s_k is not a specification of g_j

(b) A pair $(g_{j'}, k')$ is a member of $E^j(\underline{s},\underline{g})$ if, for some j,k

(1) $s_k \subset g_j$

(2) g_j is a variable symbol

(3) $(g_{j'}, k') \in E(\underline{s},\underline{B}(g_j, s(k), \underline{g}))$

__Example__: In comparing

$\underline{g} = \underline{+}\ \underline{V_1V_1}$

$\underline{s} = \underline{+}\ \underline{ab}$,

$E(\underline{s},\underline{g})$ is formed in the following steps.

(a) The only non-variable symbol in g is $+$. The corresponding symbol in $\underline{s}$ is also $+$, hence $E^*(s,g) = \emptyset$, the null set.

(b) Symbol g_1 is the variable symbol V_1. The corresponding symbol in $\underline{s}$ is $\underline{a}$. Making the substitution

$$B(g_1, s(1), g) = g^1 = +\ aa$$

we find $E^1(s,g) = E(s,g^1)$.

(b.1) By comparison of corresponding symbols, the only mismatch which occurs is at position 2, where $\underline{g}_{\underline{2}}^1 = a$, $\underline{s}_{\underline{2}} = b$. Therefore,

$$E*(s,g^1) = \{(a,2)\}$$

(b.2) String $\underline{g}^1$ contains no variable symbols, so $E^j(s, g^1) = \emptyset$ for all j , hence

$$E(s, g^1) = E*(s, g^1) = E^1(s,g) = \{(a,2)\}$$

(c) Now consider $g_2 = V_1$, the second variable symbol in g . By repeating the above steps, except that we are now concerned with g_2 instead of g_1 , we find that

$$E^2(s,g) = E(s,g^2) = \{(b,1)\}$$

(d) Combining the above steps

$$E(s,g) = \{(a,2), (b,1)\}$$

<u>Comment</u>: Difference sets between two strings are defined by syntactic considerations only. An advantage of this is that we can use the same algorithm to generate difference sets for any deductive system which fits our formalism. This provides us with a very general theorem prover, which requires little guidance from the person using it. On the other hand, this definition does not include all differences which a person might find between two strings.

5.2 <u>Problems</u>: Given two strings <u>s</u> and <u>g</u> , (the <u>starting</u> <u>state</u> and the <u>goal</u> <u>state</u>, respectively) a <u>problem</u> is defined as the location of a sequence of operator applications which demonstrate that <u>s</u> S <u>g</u>. Symbolically, a problem is solved when an ordered set of operators $\{0_{i_{\underline{a}}j_{\underline{a}}}\}$, $\underline{a} = 1\ldots\underline{n}$ is found such that

$$0_{i_nj_n}(0_{i_{n-1}j_{n-1}}\ldots(0_{i_aj_a}\ldots(0_{i_1j_1}(\underline{s})\ldots)\ldots)) = \underline{s}^n$$

and

$\underline{s}^n$ is equivalent to $\underline{g}$. When $\underline{s}^n$ is produced, $E(\underline{s}^n, \underline{g})$ will be empty.

The step $O_{i_a j_a}(\underline{s}^{a-1})$ can be a member of the sequence of steps in a proof only if $\underline{s}^{a-1}(j_a)$ is a specification of I_{i_a}. In section 4.5 we noted that the right hand string of a rewriting rule described its results. By the above argument, we now note that the left hand string establishes conditions upon a structure to which the rewriting rule can be applied. These conditions are defined collectively by saying that the structure must be a specification of the left hand string of the rule. If $E(\underline{s}(j), I_i)$ is empty, then the operation $O_{ij}(\underline{s})$ is defined. If the operation is not defined, but if a sequence of operators can be found which changes $\underline{s}$ into a string $\underline{s^*}$ such that $O_{ij}(\underline{s^*})$ is defined, then that sequence may be applied to $\underline{s}$, and the final operator, O_{ij}, to $\underline{s^*}$. Finding the sequence leading from $\underline{s}$ to $\underline{s^*}$ is itself a problem, so a problem contains within itself subproblems which can be attacked by the same methods used to solve the original problem. The argument that this is so is independent of the methods used. It is, simply, that if the subproblem is the same type of problem as the original problem, and a rational approach was used on the original problem, then why should the subproblem be handled differently?

6. Proof plans:

6.1 General: The first step in theorem proving in this formalism is to search for an operator which, if applicable, will remove elements from the difference set $E(\underline{s},\underline{g})$. Once such an operator is found, the problem of making it applicable is established and, if possible, solved. The resulting string is then examined to see if a solution has been achieved. If not, the procedure is repeated, until a solution is reached or until patience (represented by a parameter established at run time) is exhausted. If this strategy is to be successful, we must select operators which generally remove more differences than they introduce. Also, the subproblems of applying operators should be less difficult to solve than the original problem. To achieve these goals, we rely heavily upon an "abbreviated" look-ahead method, which assumes that the problem of applying operators at any one step can be solved, and then asks how much benefit will be gained from their application.

6.2 Finding a proof plan:

6.2.1 Zero-Order Operators: Let $\underline{D}^0 = E(\underline{s},\underline{g})$ be the set of zero-order differences. For every (g_j,k) in $\underline{D}^0$, a difference between $\underline{s}$ and $\underline{g}$ would be eliminated if s_k were to be rewritten as a specification of g_j. This can only be done by rewriting some substructure $\underline{s(q)}$ of $\underline{s}$ which contains s_k as one of its symbols. By examining the tables of symbol transformations, the effects of applying each rewriting rule at each position in $\underline{s}$ can be determined, by substituting $\underline{r}_i$ into $\underline{s}$ at the appropriate place. If the use of $\underline{R}_i$ on $\underline{s(q)}$ will change s_k to a specification of g_j, the operator 0_{iq} is added to the set $\underline{F}^0$ of zero-order operators. At this point, no check has been made to determine whether $0_{iq}(\underline{s})$ is defined, i.e., to see whether $\underline{s(q)}$ is a specification of $\underline{\ell}_i$.

6.2.2 First-order differences: In examining the second table above, suppose we find that some operator 0_{ij} will change s_k to s_n, but s_n is not a specification of g_j. Now, if s_n could be rewritten so that it was a specification of g_j, then 0_{iq} could be used to remove the difference (g_j, k) from $\underline{D}^0$. We have thus discovered a new interesting difference, namely

(g_j, n), and it would seem reasonable to tackle it in the same way as we did the original difference. To this end, (g_j, n) is added to the set $\underline{D}^1$ of first-order differences.

6.2.3 Higher-order differences and operators: After all the members of $\underline{D}^0$ have been examined, the set of zero-order operators and first-order differences will be complete. The set $\underline{D}^1$ of first-order differences is then examined, in exactly the same manner as $\underline{D}^0$ was examined. This generates two further sets, $\underline{P}^1$, the set of first-order operators, and $\underline{D}^2$, the set of second-order differences. Repeating the same procedure again results in $\underline{D}^3$ and $\underline{P}^2$. More generally given $\underline{D}^m$, the sets $\underline{P}^m$ and $\underline{D}^{m+1}$ can be generated. The process continues until $\underline{D}^i$ is empty, or $\underline{P}^p$ has been generated, where $\underline{p}$ is a parameter specified externally. To avoid looping, an operator or difference is not added to its appropriate set if it is already a member of that or some lower-order set.

Comment: By using this procedure we have a definition of differences between states which is entirely syntactic. The user of a program need not specify any special system specific routines for defining differences. The user can leave the definition of differences completely under program control by not defining classes of terminal symbols (e.g., by treating + and - as unique symbols, instead of including the class adop. in the algebra example.).

A truly general general theorem prover would specify the value of $\underline{p}$ internally, thus deciding to go to great depths in exploring promising solutions while abandoning unpromising lines of approach rapidly. To do this, one would have to have some way of determining the probability that a particular line of attack was going to result in a successful solution. We do not at this time have a specific proposal which we care to defend. As a practical matter, our operating program can accept an initial value of $\underline{p}$ and then, if that does not work, increase $\underline{p}$ and try the problem again.

6.3 Ordering of operators: Let $\underline{P}$ be the set of sets $\{\underline{P}^h\}$, $\underline{h} = 0 \ldots \underline{p}$. $\underline{P}$ contains all the information available from which operators can be selected as likely candidates for the innermost operator, $0_{i_1 j_1}$, in the proof expressed in 5.2. The relevant information consists of the number and level of differences which generated each operator, the complexity of the operator (obtained by examining its rewriting rule), and the number of corrections required before the operator can be applied (obtained by comparing its rewriting rule to the relevant substructure $\underline{s(i)}$ of $\underline{s}$). Let $F(\underline{P})$ be defined as a function which orders the set of operators $\{0_{ij}\}$, which appear as members of pairs in the sets $\{P^h\}$. For brevity, we will write $\underline{F}$ for the ordered set of operators. $\underline{F}$ is interpreted as an ordering, from "probably most useful" to "probably least useful", of operators which may lead to the next step in a proof.

Comment: In a specific program, the definition of $F(\underline{P})$ is crucial. In describing the structure of a class of theorem provers, it is sufficient to assert that $F(\underline{P})$ exists and that it orders operators. An advantage of the formalization is that it makes clear ways in which two programs could differ and still fall within the same class of programs. For instance, it is conceivable that in different problem areas (e.g., trigonometry as opposed to the predicate calculus) different algorithms for $F(\underline{P})$ would be appropriate. In our own work we have experimented with several algorithms. The results of these studies will be reported elsewhere.

7. Procedures for solving problems:

7.1 General: Using the above definitions, we now present two procedures for problem-solving. The first is a recursive procedure, in the "Algol" sense that a class of algorithms are specified which call themselves as subroutines. This procedure has a structure very similar to that of a top-down compiler-compiler. We then show that there exist problems which cannot be solved by any algorithm in the class. Next we described a modified, non-recursive procedure which describes the structure of our current working programs. Henceforth this will be referred to (for historic reasons only) as the "Fortran Deductive System", or FDS, algorithm. We show that problems which previously escaped solution using the recursive procedure can be captured by the FDS algorithm.

7.2 The recursive procedure: Given the problem of rewriting s as a specification of g, and two externally specified parameters r and p, proceed as follows:

(1) Set $k = 0$, $s^* = s$, $g^* = g$. (The variable k will be used to simulate the level of recursion of the problem-solving procedure).

(2) Set $k = k + 1$. If $k > r$, then go to step (7) , otherwise go to step (3) .

(3) Determine $D^0 = E(s^k, g^k)$, then P (section 6) , and set $F^k = F(P)$. If $D^0 \neq \emptyset$ go to step (6); otherwise assert that the problem s^k , g^k is solved, set $\hat{k} = k$, $k = 0$ and go to step (4) .

(4) Set $k = k + 1$. If $g^k = g^{\hat{k}}$ then go to step (5) , otherwise return to step (4) .

(5) [illegible]

(5) Set $\underline{k} = k - 1$. If $\underline{k} = 0$, assert that the main problem is solved, and quit. Otherwise, find the operator 0_{ij} which generated the goal $\underline{g}^k$, set $\underline{s}^{k+1} = 0_{ij}(\underline{s}^k)$, $\underline{g}^{k+1} = \underline{g}^k$, $\underline{p} = \underline{p} + 1$, and go to step (2) .

(6) If $\underline{F}^k = \emptyset$ then go to step (7) , otherwise go to step (8) .

(7) The problem $\underline{s}^k$, $\underline{g}^k$ cannot be solved. If $\underline{k} = 1$, then assert total failure and quit; otherwise set $\underline{k} = \underline{k} - 1$. If $\underline{g}^k \neq \underline{g}^{k+1}$, then set $\underline{p} = \underline{p} + 1$. Go to step (6) .

(8) Remove the first operator from $\underline{F}^k$; call it 0_{ij} . If $0_{ij}(\underline{s}^k)$ is defined, set $\underline{s}^{k+1} = 0_{ij}(\underline{s}^k)$, $\underline{g}^{k+1} = \underline{g}^k$ and go to step (2) . Otherwise, if $\underline{p} > 0$, go to step (9) else return to step (6) .

(9) The subgoal of applying 0_{ij} is to be established. Let $f(\underline{i})$ be a function whose value is $\{\underline{\alpha},\underline{\beta}\}$, where $\underline{s^k(i)}$ is the $\underline{\alpha}$th direct component of s^k_β . Initially, set $\underline{g^{k+1}}$ to the string $\underline{I_i}$, then execute the following steps in sequence:

(9.1) If $\underline{i} = 0$, go to step (9.5).

(9.2) Let $f(\underline{i})$ be $\{\underline{\alpha},\underline{\beta}\}$. Set $\underline{i} = \underline{\beta}$.

(9.3) Let the degree of symbol $s^k_{\underline{i}}$ be $\underline{n}$, and let $\underline{X_n}$ mean "any symbol of degree $\underline{n}$" . Form the string $\underline{u} = \underline{X_n}\ \underline{V_{i+1}}\ \underline{V_{i+2}} \cdots \underline{V_{i+\alpha-1}}\ g^{k+1}\ \underline{V_{i+\alpha+1}} \cdots \underline{V_{i+n}}$ where $\underline{i}$ is defined as the highest index of a free variable in g^{k+1} . Set g^{k+1} to $\underline{u}$.

(9.4) Return to step (9.1) .

(9.5) Set $\underline{p} = p - 1$, $\underline{s}^{k+1} = \underline{s}^{k}$, and go to step (2) .

<u>Comment</u>: After steps (9.1) - (9.5) have been executed and control returns to step (2), $\underline{g}^{k+1}$ will be a structure with the following properties:

(a) The symbol s_j^k in $\underline{s}^k$ will correspond to the symbol marking the start of the structure $\underline{X_i}$ in g^{k+1} .

(b) The first symbol of every other structure in $\underline{s}^k$ will correspond to a free variable symbol in $\underline{g}^{k+1}$. Thus, if $s^k(j)$ is a specification of X_i , $\underline{s}^{k+1}$ is guaranteed to be a specification of $\underline{g}^{k+1}$.

If the procedure finds a solution, it will produce one using not more than $\underline{r}$ levels of recursion. At no state will it "look ahead" more than $\underline{p}$ subproblems to see if an operator may be applied. Interestingly, increasing the values of $\underline{r}$ and $\underline{p}$ beyond the minimum values needed to find any proof hardly ever results in finding a more direct proof. This is because the increased parameters will permit an extension of the set $\underline{F}$ to more operators, and unless the ordering within $\underline{F}$ is perfect, there is always a chance that an operator which leads to no proof, or to a true but awkward proof, will be tried before an operator which leads to a direct proof. Some examples of this phenomonon are given in (9).

<u>Comment on programming</u>: We have presented a non-recursive method of storage allocation by manipulating the pointer $\underline{k}$ and indexing $\underline{s}$, $\underline{g}$ and $\underline{F}$. In a programming language which permits dynamic storage allocation, the coding problem is much simpler.

7.3 <u>A class of problems not handled by the recursive procedure</u>: Starting with an original problem $\underline{s}^1$, $\underline{g}^1$, suppose that the subproblem

$\underline{s}^2$, $\underline{g}^2$ is generated, as in step (9) of 7.2. Any rewriting of $\underline{s}^2$ into a structure which is a specification of $\underline{g}^2$ is a solution to the subproblem. Let us suppose that two such solutions exist resulting in structures $\underline{s}^*$ and $\underline{s}^{\#}$. By hypothesis, the rewriting rules given could generate either of them. A particular ordering function, however, will uncover one of them first. Let this be $\underline{s}^*$. Upon finding this rewriting, $\underline{s}^3$ will be defined as

$$\underline{s}^3 = O_{ij}\ (\underline{s}^*)\ ,$$

and control will be passed to step (2) at which point the problem $\underline{s}^3$, $\underline{g}^3$ is attempted. Assume that this problem cannot be solved. Control will be returned to step (6) of 7.2. At this point, if O_{ij} was the last operator on $\underline{F}^1$, $\underline{F}^1$ will now be empty and failure will be asserted. Failure will also be asserted, although somewhat later, if the operators following O_{ij} on $\underline{F}^1$ do not produce steps which lead to a proof.

Suppose, however, that the step $O_{ij}(\underline{s}^{\#})$ was an irreplaceable step in the proof. By hypothesis, the rewriting rules exist such that $\underline{s}^2$ can be rewritten as $\underline{s}^{\#}$, but in the formulation described in 7.2, this step will never be reached since once $\underline{s}^*$ is found, as O_{ij} will be removed from $\underline{F}^1$. On the other hand, not removing operators from $\underline{F}^1$ would result in looping. In brief, any solution to a subproblem is assumed unique.

7.4 An example of the class: The following example illustrates both the procedure of 7.2 and the flaw described in 7.3.

The problem area is a greatly reduced subset of algebra. Let the vocabulary consist of

(1) The terminal symbols X and 0 , of degree 0 , and $+$ of degree 2.

(2) Free variables V_1 as necessary.

(3) The terminal classes $C_1 = \{+\}$, $C_2 = \{X\}$ and $C_3 = \{0\}$.

The set R is the three rewriting rules

$$R_1 :\equiv + 0 V_1 := V_1$$

$$R_2 :\equiv V_1 := + 0 V_1$$

$$R_3 :\equiv + V_1 V_2 := + V_2 V_1$$

The initial problem is

$$s = + X 0 , g = X$$

Setting s^1 and g^1 to these strings, step (3) of 7.2 generates $D^0 = \{(X, 0)\}$, from which we get $F^1 = \{0_{1,0}\}$. Since $0_{1,0}(+ X 0)$ is not defined, we generate the subproblem

$$s^2 = + X 0 , g^2 = + 0 V_1$$

The subproblem generates the set of zero order differences, $D^0 = \{(0,1), (V_1, 2)\}$, and $F^2 = \{0_{2,0}, 0_{3,0}\}$. Since $0_{2,0}(s^2)$ is defined, the new problem

$$s^3 = + 0 + X 0 , g^3 = + 0 V_1$$

is generated. s^3 is now a specification of g^3 , so by repeating step (4) set k to 1 and, after using $0_{1,0}$ on s^3 , continue with

$$s^2 = + X 0 , g^2 = X .$$

Now $s^2 = s^1$ and $g^2 = g^1$; the problem-solving procedure is caught in a loop. This situation is detected, the problem s^2, g^2 is failed, and the system returns to the situation

$\underline{k} = 1$, $\underline{s}^1 = \underline{+\ X\ 0}$, $\underline{g}^1 = \underline{X}$, $\underline{F}^1 = \emptyset$

from which step (7) of 7.2 asserts failure.

7.5 <u>The non-recursive modification</u>: There were two solutions to the problem $\underline{s}^2$, $\underline{g}^2$, and the program accepted the wrong one. In the modification, the fully recursive structure is abandoned, allowing the program to try a subproblem a second time if necessary.

This is done by not "reporting back" when a subproblem is solved, but rather bringing forward the previous goal. Specifically, if while considering problem $\underline{s}^i$, $\underline{g}^i$, the operator $0_{\alpha\beta}$ generates the subproblem $\underline{s}^{i+1} = \underline{s}^i$, $\underline{g}^{i+1}$, and this is solved by finding a structure $\underline{s}^{i+j}$ which is a specification of $\underline{g}^{i+j} = \underline{g}^{i+1}$, then the new subproblem $\underline{s}^{i+j} = 0_{\alpha\beta}(\underline{s}^{i+j})$, $\underline{g}^{i+j} = \underline{g}^i$ is established. From its index, this is seen to be a subproblem of $\underline{s}^{i+j-1}$, $\underline{g}^{i+j-1}$. (In the case $\underline{g}^{i+1} = \underline{g}^1$, success is asserted.) Note that if the particular subproblem $\underline{s}^{i+j}$, $\underline{g}^{i+j}$ turns out to be unsatisfactory, a new solution will be sought for $\underline{s}^{i+j-1}$, $\underline{g}^{i+j-1}$.

To achieve this, change step (5) of 7.2 .

(5) If $\underline{g}^k = \underline{g}^1$, assert that the main problem is solved and quit. Otherwise, set $\underline{k} = \underline{k} - 1$. Find the operator 0_{ij} which generated $\underline{g}^p$, set $\underline{s}^k = 0_{ij}(\underline{s}^k)$, $\underline{g}^k = \underline{g}^k$, $\underline{p} = p + 1$, $\underline{k} = \underline{k} - 1$ and go to step (2) .

<u>Comment</u>: Clearly, this destroys the recursive nature of the algorithm, since it permits high level subproblems to have access to low level goals.

7.6 <u>The example problem revisited</u>: The modified procedure solves the illustration problem as follows:

The development is identical to 7.4 through the first occurrence of the problem $\underline{s}^3$, $\underline{g}^3$. Following its solution, instead of returning to $\underline{s}^2$, $\underline{g}^2$, we have

$$\underline{s}^3 = \underline{+}\ \underline{X}\ \underline{0}\ ,\ \underline{g}^3 = \underline{X}$$

Again we note $\underline{s}^3 = \underline{s}^1$ $\underline{g}^3 = \underline{g}^1$; so a return is made to the situation

$$\underline{k} = 2\ ,\ \underline{s}^2 = \underline{+}\ \underline{X}\ \underline{0}\ ,\ \underline{g}^2 = \underline{+}\ \underline{0}\ \underline{V_1}\quad \underline{F}^2 = \{0_{3,0}\}.$$

Since $0_{3,0}\ (\underline{s}^2)$ is defined, we get the new problem

$$\underline{s}^3 = \underline{+}\ \underline{0}\ \underline{X}\ ,\ \underline{g}^3 = \underline{+}\ \underline{0}\ \underline{V_1}$$

$\underline{s}^3$ is a specification of $\underline{g}^3$, and so by step (5) we have

$$\underline{s}^3 = 0_{1,0}(\underline{+}\ \underline{0}\ \underline{X}) = \underline{X}\ ,\ \underline{g}^3 = \underline{X}$$

Now, $\underline{s}^3$ is a specification of $\underline{g}^3$, and since $\underline{g}^3 = \underline{g}^1$ the original problem is solved.

<u>Comment</u>: The modified procedure has proven that $\underline{0}\ \underline{+}\ \underline{X} = \underline{X}$, hardly a result of great surprise. And it did it with only one false start! "Obviously", one would avoid the need for elaborate procedural arrangements of this sort if the correct operator was tried first. If an algorithm for doing this were known, the act of proving theorems would be trivial. The modified procedure safeguards one against less than perfect orderings of operators.

8. <u>Discovery of solutions to problems</u>: We shall now define the class of problems which our modified system can solve. For this purpose, we shall assume that the parameters $\underline{r}$ and $\underline{p}$ are unbounded, as limiting these serves merely to increase the efficiency of the system.

8.1 <u>Notation</u>: Let $\overset{n}{\underset{\alpha=m}{Y}} 0_{i_\alpha j_\alpha}$ denote the sequence of operations defined by

$$\overset{n}{\underset{\alpha=m}{Y}} 0_{i_\alpha j_\alpha}(\underline{s}) = 0_{i_n j_n}(0_{i_{n-1} j_{n-1}}(\ldots 0_{i_m j_m}(\underline{s})\ldots)) .$$

This compound operator will be used only where each operation in the sequence is defined. If $\overset{n}{\underset{\alpha=m}{Y}} 0_{i_\alpha j_\alpha}(\underline{s})$ is a specification of some structure $\underline{g}$, then the ordered set $\{0_{i_\alpha j_\alpha}, \alpha = \underline{m}, \underline{n}\}$ is a solution to the problem $\underline{s}$, $\underline{g}$. Finally, let $G(0_{ij}(\underline{s}))$ denote the "try to apply 0_{ij} to $\underline{s}$" . The generation of an appropriate goal structure was defined in 7.4.9 .

8.2 <u>Difference-discoverable solutions</u>: In general a problem has many possible solutions. These can be dichotonized by asking the following question: At each step $\underline{s}^k$, $\underline{g}^k$ in the solution, can what to do next be discovered by analysis of only $\underline{s}^k$ and $E(\underline{s}^k, \underline{g}^k)$? Formally, we call a solution $\pi = \{0_{i_\alpha j_\alpha}, \alpha = \underline{m},\underline{n}\}$ to the problem $\underline{s}$, $\underline{g}$ "difference-discoverable" if

8.2.1 $\underline{s}$ S $\underline{g}$ and $\pi = \emptyset$, or

8.2.2 there exists an $0_{i_\beta j_\beta}$ contained in $\pi \cap \underline{P}$ such that the solutions $\{0_{i_\alpha j_\alpha}, \alpha = \underline{m}, \underset{\beta}{}\beta - 1\}$ to $\underline{s}$, $G(0_{i_\beta j_\beta}(\underline{s}))$ and $\{0_{i_\alpha j_\alpha}, \alpha = \beta + 1, \underline{n}\}$ to $\overset{}{\underset{\alpha=m}{Y}} 0_{i_\alpha j_\alpha}(\underline{s})$, $\underline{g}$ are both difference-discoverable.

<u>Comment</u>: $\underline{P}$ is constructed in such a way that it will contain every operator which reduces $\underline{D}$, and hence may be instrumental in reducing $\underline{D}^0$. Thus, $0_{i_\beta j_\beta} \epsilon \underline{P}$ implies that an attempt will be made to form $0_{i_\beta j_\beta}(\underline{s})$. If $0_{i_\beta j_\beta}$ is also a member of π , then this attempt and the subsequent application of $0_{i_\beta j_\beta}$ will, if made, lead to the first β steps of π

being discovered. (The attempt may not be made if the βth step of π is reached in some other way.)

Example: In understanding this definition, it is illuminating to consider an example of a solution which is not difference-discoverable. Let $\underline{\text{MINUS}}$ denote unary minus, and define the rewriting rules

$$\underline{R_1}: \equiv \quad \underline{\text{MINUS}}\ \underline{x}\ \underline{V_1}\ \underline{V_2} := \underline{x}\ \underline{V_2}\ \underline{\text{MINUS}}\ \underline{V_1}$$

$$\underline{R_2}: \equiv \quad \underline{x}\ \underline{V_1}\ \underline{V_2} := \underline{x}\ \underline{V_2}\ \underline{V_1}$$

Consider the problem

$$\underline{s} = \underline{\text{MINUS}}\ \underline{x}\ \underline{a}\ \underline{b},\ \underline{g} = \underline{x}\ \underline{a}\ \underline{\text{MINUS}}\ \underline{b}$$

which has the solution

$$\pi = \{0_{2,1}, 0_{1,0}\}$$

$\underline{D}^0 \equiv E(\underline{s},\underline{g}) = \{(\underline{x}, 0)\}$, $\underline{P} = \{0_{1,0}\} = \pi \cap \underline{P}$. Since $\underline{s}\ \$\ \underline{g}$, the only $0_{i_{\beta-\beta}}$ for 8.2.2 is $0_{1,0}$. Thus the solution π to $\underline{s}$, $\underline{g}$ is difference-discoverable if and only if the solution $\pi^* = \{0_{2,1}\}$ to $\underline{s}$, $\underline{g}^* = G(0_{1,0}(\underline{s}))$ and $\pi^{**} = \emptyset$ to $\underline{s}^{**} = 0_{1,0}(0_{2,1}(\underline{s}))$, $\underline{g}$ are difference-discoverable. But $\underline{s}\ S\ \underline{g}^*$ and $\pi^* \neq \emptyset$, so the solution π^* to $\underline{s}$, $\underline{g}^*$ is not difference discoverable; thus, neither is the solution π to $\underline{s}$, $\underline{g}$.

The result could be predicted from the intuitive concept of difference-discoverability given in the beginning of this section. Initially, $\underline{D}^0 = \{(\underline{x},0)\}$. The only way to remove this difference is to use $\underline{R_1}$. However, the fact that $0_{2,1}$ must be used to switch $\underline{a}$ and $\underline{b}$ before removing this difference cannot be detected by the procedures given in section 6.2.

8.3 Problem-solving scope of the system: Any difference-discoverable solution to a problem can be found by the FDS algorithm. Since we shall be talking about the system finding a particular solution, we add (temporarily) an extra switch, or "demon" to the system. This will check any solution found, and if it is not the required one, will discard it and keep looking.

Let $\underline{s}$, $\underline{g}$ be a problem with the required difference-discoverable solution $\pi = \{0_{i_\alpha j_\alpha}, \alpha = 1,\underline{n}\}$. Our proof will be by induction on $\underline{n}$, the number of steps in the required solution.

In the case $\underline{n} = 0$, the theorem is trivial; $\underline{s} \mathrel{S} \underline{g}$ and since π is difference-discoverable, $\pi = \emptyset$ by 8.2.1. The required proof will be detected by step (3) of 7.2.

Assume then, that any difference-discoverable solution of less than $\underline{n}$ steps can be found. Since $\underline{n} > 0$, $\pi \neq \emptyset$, and so there must be an $0_{i_\beta j_\beta}$, $1 \leq \beta \leq \underline{n}$ which satisfies 8.2.2. That is, the required solutions $\pi^* = \{0_{\beta i_\alpha j_\alpha}, \alpha = 1, \beta - 1\}$ to $\underline{s}$, $G(0_{i_\beta j_\beta}(\underline{s}))$ and $\pi^{**} = \{0_{i_\alpha j_\alpha}, \alpha = \beta + 1, \underline{n}\}$ to $\underset{\alpha=1}{\Upsilon} 0_{i_\alpha j_\alpha}(\underline{s})$, $\underline{g}$ are difference-discoverable. Since these are both of $\underline{n}$ steps, they can be discovered. The demon guarantees that eventually all difference-discoverable solutions, and hence π , will be discovered.

Comment: To see why this proof does not apply to the recursive formulation, add the same demon. Now suppose that the first solution found to $\underline{s}$, $G(0_{i_\beta j_\beta}(\underline{s}))$ is not π^* . When failure is finally asserted on the other problem, $0_{i_\beta j_\beta}$ will be removed from $\underline{P}$, and no further solution sought to the first subproblem. Thus, the discovery of π^* cannot be guaranteed, so neither can discovery of π .

If a problem has at least one difference-discoverable solution, a solution will be found. The particular one found will depend on the ordering function F(P) . Empirically, we have found that problems without at least one difference-discoverable solution are rare. Only one such problem has been found in our study of over one thousand problems from diverse areas of mathematics. Thus, we feel that this limitation is not overly restrictive.

It would be nice if there were some way to look at a problem and say that it does, or does not, have a difference-discoverable solution. This is a chimera. Difference-discoverability is a property of solutions, not of problems. To state in advance that a problem does or does not have a difference-discoverable solution requires that one make an assertion about the form of a solution before he knows what the solution is.

Another way of looking at this is to say that we are dealing with an algorithm which is not complete - since it will not always prove a true theorem. This is in contrast to theorem provers based on the resolution principle, for which completeness can be proven (10). Should we be interested in incomplete theorem proving methods? The answer depends on their pragmatic utility. Is the expense of possible failure offset by an ability to handle some problems very well? The following section addresses this point.

9. Examples

9.1 General: We will now show, by illustration, where the FDS algorithm does or does not perform well. The algorithm is quite effective if the theorem has a proof in which each line is derived by application of a rule of inference to the line derived immediately before it. This will be referred to as the "line by line property." The algorithm is much less suited, and indeed, appears clumsy, if the "natural" proof (to a human) depends on the convergence of two or more chai[illegible] of inference. Because many very difficult proofs are of the latter type, and because we have used very simple examples, it may appear at first that the algori[illegible]m is suitable only for trivial problems. This is not so, as a line by line proof may involve steps which are difficult to find, and our simple examples were chosen solely for ease of exposition. Several of the following illustrative pr[illegible]lem[illegible] are quite difficult for people. As appropriate, we shall cite st[illegible]tistics.

Within each example, a program fitting the FDS algorithm (i.e., behaving as we have specified except tha[illegible] a specific ordering rule was stated to define F(P)) was given an initial set of rewriting rules, then a series of problems. Each time a problem was solved, (except in pattern identification) the appropriate theorem was added to the list of rewriting rules available for subsequent problems.

9.2 Elementary algebra: Elementary algebra is a system in which most proofs have the line by line property, and hence the algorithm works well. This is illustrated in Tables 1 and 2, which summarize the result of an experiment with school algebra problems. Table 1 states a set of axioms and problems involving algebraic manipulation of plus and binary minus. In Table 2 manipulations of zero and unary minus are defined.

"Everyone knows" these theorems are true, but it is [illegible]urprisingly hard to prov[illegible] them. The rewriting rules and problems one to thirteen of Table 1 were presented to fifteen University of Washington undergraduates enro[illegible] in introductory psychology. Although high school algebra is a requirement for admission to the University, and [illegible]entary statistics taught as part of this course, the students did quite poorly. They averaged two and a half solutions in half an hour. The best problem solver produced only seven proofs.

Some of these problems are not trivial for people who have had considerable formal mathematical training. In particular, consider problem 7 of Table 1. We have, informally, presented this problem to about a dozen graduate students or Ph.D.'s in Mathematics, Engineering, and Computer Science, allowing from ten to forty-five minutes to work on it. Only a few people solved the problem. The fastest solution obtained, a different proof from that found by FDS, was found in ten minutes by an undergraduate, who subsequently was elected to Phi Beta Kappa in Mathematics for other reasons.

9.3 Sanderson algebra: "Sanderson algebra" is an algebra developed by Dr. J. Sanderson to describe simple flow charts, such as might occur in microprogramming, (12). Proving that two well formed expressions are equivalent in this algebra proves that the flow charts they describe are computationally equivalent. Table 3 lists the twenty-six "axioms", or equivalences, which Sanderson proved by showing that all possible inputs to the corresponding circuits gave equivalent outputs. He then proved the fifteen theorems listed as problems in the table. The theorems have also been proven by the FDS, with the times listed.

This example illustrates the strong "non-reflectivity" of the FDS algorithm. The algorithm must begin with the starting state, and then rewrite to the goal state, where a human theorem prover might reverse the process. This is illustrated in Table 3, problems 13 and 14. These are reflexive versions of each other, yet by a time to solution criterion, one problem is 500 times harder than the other! In practice, the FDS algorithm ought to be supplemented by a pre-processor which decided that *if* ":=" was reflective in the deductive system being handled, and the problem was of the form "a := b" , it would be easier to prove "b := a". But how is this decision to be made?

Eight University of Washington Junior or Senior majors in Mathematics spent three hours each attempting to prove the problems of Table 3. The system was presented to them with reflectivity of ":=" assumed, so that if anything, they had an easier task than did the FDS program. (There are then seven problems.) The average student obtained two solutions. Only one student solved all problems in eighty minutes.

9.3 Pattern identification: Our next example involves parsing strings in a phrase structure grammar. This example was chosen to show the resemblelance of the FDS algorithm to compilation. In addition, this example illustrates the application of the algorithm to a slight extension of the theorem proving problem. Instead of being asked to find a sequence of transformations which change the starting state into the goal state, the program is asked to detect to which of several classes of strings a particular well formed expression belongs.

Two dimensional line drawings of buildings provide the subject matter. For ease of exposition, think of "houses" and "churches." Prototype houses and churches are shown in Figure 1. Given a set of basic patterns, complex two dimensional variations of these patterns can be constructed by specifying a set of connectives (e.g., "on top of") and a phrase structure grammar whose rules determine the variation of patterns which fall into a particular class, (4, 5). A typical set of rules are given in (4) . These are similar to the rules used in this application, with the following exception. Let $\{B_i\}$, $i = 1...n$, be the set of $\underline{n}$ prototypes, in this case the $\underline{n}$ basic patterns representing different types of "house" or "church". The rewriting rules contain $\underline{n}$ rules of the form

$$B_i := \text{PATTERN}$$

where "PATTERN" is a special constant that only appears in rewriting rules on the right hand side of the above $\underline{n}$ special rules and in the rule PATTERN . PATTERN := PATTERN . If the problem A := PATTERN is presented, where A is any well formed expression, if the program solves this problem, it will have first rewritten A as some collection of the $\{B_i\}$ then made the trivial last steps of rewritting to PATTERN. If the B_i each represent one of the basic types of patterns, then the well formed expression (complex picture) A will have been classified as a particular type of pattern.

Figure 2 shows two basic patterns and one of the compound patterns which have been classified as examples of each type. In this example there are eight basic patterns, and 26 problems were attempted. Proofs in this application are quite direct, and are obtained in an average time of six seconds.

9.5 Propositional calculus: The previous illustrations have been of theorem proving in systems which have the line by line property. More specifically, each system has axioms of the form X is equivalent to Y , with the added assumption that within an expression any subexpression may be replaced by one of its equivalent expressions. We will now give an example of how poorly the FDS algorithm performs if it is asked to prove theorems in a system which does not have this property. The system is the propositional calculus. The following axioms are true statements, but not statements of equivalences.

$$a \supset (b \supset a)$$

$$(a \supset (b \supset c)) \supset ((a \supset b) \supset (a \supset c))$$

$$(\sim b \supset \sim a) \supset ((\sim b \supset a) \supset b)$$

To begin an FDS computation at all, these were expressed by three rewriting rules of the form

$$x := x,(a \supset (b \supset a))$$

together with the rule of inference

$$(x,a),\ (a \supset b) \quad := \quad x,b \quad .$$

The first set of rules say, in effect, that any expression may be rewritten as itself and an axiomatic statement. The second is a rewriting form of modus ponens. In ten minutes the system was unable to prove that $a \supset a$, is a true statement. The reason is that in any system where one step is required for each substitution, this is an extremely long proof. Also, the starting state provides very little clue as to where to begin. This is seen by examining the rewriting rule form of the problem

$$x := x,\ a \supset a \quad .$$

The algorithm is appropriate for many rewriting problems which appear to be like the propositional calculus, but are in fact problems of inference. A systematic examination of the problem sections of Suppes and Hill's Introductory Logic for the Schools (14), uncovered no problems which, in principle, could not be solved by the FDS method. In some cases we did not care to expend the necessary computing time on hard examples. Table 4 illustrates system performance on problems Suppes and Hill used to illustrate inferences by modus tollens, modus ponens, and double negation. Our observation was that most of the applications of logic in this grade school book were to the use of logic to draw inferences, rather than to prove theorems.

9.6 A puzzle: This example is a version of a well known logic puzzle. Most people find it hard the first time they see it.

There are two protagonists, Ed and Al, each of whom either always tells the truth or always lies. A philosopher approaches the pair and asks if the library is to the East or West. Ed mutters something unintelligible, and Al states clearly, "Ed says east, but he's a liar." Where is the library?

FDS uses the notation

FDS Expression	Meaning
SAYS (A/B)	A makes statement B
A.IS.TTLE	A is a truthteller (i.e., honest citizen)
A.IS.LIAR	A is a liar (i.e. gangster)
A.IM.B	A implies B
A.EQ.B	A is equivalent to B
DIRN	A direction, i.e. either east or west
A'.D'B	A may be rewritten as B
(A)AND.B	A and B

and the rewriting rules:

```
SAYS(A/B)   := (A.IS.TTLR).EQ.B
A.IS.LIAR   := NOT (A.IS.TTLR)
NOT(EAST)   := WEST
NOT(WEST)   := EAST
(A.EQ.B) AND.NOT (B)   := DATA.IS.NOT (A)
A.EQ.B   := B.EQ.A
A.EQ. (B.EQ.C)   := (A.EQ.B) EQ.C
A.EQ. NOT(B)   := NOT (A.EQ.B)
```

The problem was presented as

```
SAYS (AL/SAYS (ED/EAST)) . SAYS (AL/(ED.IS.LIAR)) := DATA.IM.DIRN
```

The program produced the following proof in 14 seconds.

```
(SAYS(AL/SAYS(ED/EAST))AND.SAYS(AL/ED.IS.LIAR)))
(((AL.IS.TTLR).EQ.SAYS(ED/EAST))AND.SAYS(AL/(ED.IS.LIAR)))
(((AL.IS.TTLR).EQ.SAYS(ED/EAST))AND.((AL.IS.TTLR).EQ.(ED.IS.LIAR)))
(((AL.IS.TTLR).EQ.SAYS(ED/EAST))AND.((AL.IS.TTLR).EQ.NOT(ED.IS.TTLR)))
(((AL.IS.TTLR).EQ.SAYS(ED/EAST))AND.NOT((AL.IS.TTLR).EQ.(ED.IS.TTLR)))
(((AL.IS.TTLR).EQ.((ED.IS.TTLR).EQ.EAST))AND.NOT((AL.IS.TTLR).EQ.(ED.IS.TTLR)))
((((AL.IS.TTLR).EQ.(ED.IS.TTLR)).EQ.EAST)AND.NOT((AL.IS.TTLR).EQ.(ED.IS.TTLR)))
((EAST.EQ.((AL.IS.TTLR).EQ.(ED.IS.TTLR)))AND.NOT((AL.IS.TTLR).EQ.(ED.IS.TTLR)))
(DATA.IM.NOT(EAST))
(DATA.IM.WEST)
```

This example illustrates a use of classes of terminal symbols. The symbol DIRN is a member of the constant class {DIRN, EAST, WEST}. Therefore, the last line in the proof is a specification of the goal state DATA.IM.DIRN , but is not identical to it.

9.7 Inequalities: The final example is the solution of introductory problem in the calculus of inequalities. Inequalities present an interesting area of mathematics in which to study theorem proving, since inequalities proofs frequently depend upon combinations of arguments from algebra and from mathematical logic. Therefore, in order to do inequalities the program must have available rewriting rules from the more basic fields. Typically only a few rules will be used in any one proof, but since their identity cannot be known before starting on a problem, all must be available. The presence of many rules will greatly increase the probability of the program's starting down fruitless paths. This consideration applies to both state to state theorem provers and to the resolution principle programs (15). Therefore, inequalities provide tests of the extent to which the generation of difference sets will provide a framework to test the selectivity of the program.

In this example the FDS program used the simple ordering procedure of determining the probable usefulness of a rewriting rule by a weighted sum of (a) the number of conditions on the string required by the rewriting rule, i.e., the complexity of the left hand side, and (b) the number of these conditions satisfied by the current state string. The system was initialized by loading rewriting rules based on algebra and the propositional calculus, including definitions of "implies", "conjunction", and "equivalance", and the axioms defining permissible algebraic manipulations with $+$, $-$, $/$, and $\times$. In addition, about one hundred previously proved theorems in both areas were loaded as rewriting rules. Table 5 lists the definitions of inequalities and the theorems proven.

The proof of the following theorem is illustrative. It was obtained at a time at which the system had to select rewritings from a set of 134 rules. The problem is to show that

$$\mathrm{REAL}\ (A) \cdot \mathrm{REAL}\ (B) := ((A > B)\ \bar{\vee}\ (A = B))\ \bar{\vee}\ (B > A)$$

is a legal rewriting. The proof is

$$\begin{aligned}
\mathrm{REAL}\ (A) \cdot \mathrm{REAL}\ (B) &:= \mathrm{REAL}\ (A - B) \\
&:= (((A - B) > 0)\ \bar{\vee}\ ((A - B) = 0))\ \bar{\vee}\ (\mathrm{NEG}(A-B) > 0) \\
&:= (((A-B) > 0)\ \bar{\vee}\ ((A-B) = 0))\ \bar{\vee}\ (B > A) \\
&= (((A-B) > 0) \supset (A = B))\ \bar{\vee}\ (B > A) \\
&= ((A > B)\ \bar{\vee}\ (A = B))\ \bar{\vee}\ (B > A)
\end{aligned}$$

The proof was produced in two minutes and thirty three seconds. Twenty three structures were considered and six used in the proof. Most of the time was spent establishing the order in which operators were to be applied. As can be seen, a good ordering was achieved.

9.8 Summary of the examples: These examples have been taken from about one thousand theorem proving problems which have been solved by different programs fitting within the framework of the FDS algorithm. The performance of a particular program on a given problem will depend upon the parameters used to limit its search and upon the rule which it uses to determine the sequence in which operators will be tried. The programs are relatively insensitive to parameter changes, but highly sensitive to the effect of different ordering rules. An interesting question, which we are now exploring, is whether or not it is possible to write a program which develops its own ordering rules.

The FDS algorithm has not produced any proofs of theorems which would be considered "deep" by a pure mathematician. On the other hand, it more than holds its own with upper division undergraduates, providing that the proofs which it is trying to discover have the line by line property. This is no mean accomplishment, given the current state of artificial intelligence research. It seems fair to say that FDS programs prove theorems as well as the better currently available chess-playing programs play chess, (3), but that it does not prove theorems as well as checkers playing programs play checkers (11).

10. Comparisons and Conclusions:

10.1 Comparison to GPS: As we have indicated, the FDS is an intellectual descendant of the GPS. (7,8). Both programs are theorem provers which solve problems by successive rewritings, from a starting state to a goal state.

There are two major differences between the GPS and FDS programs. The most obvious, and most trivial, is that the GPS is a list processing program written in IPL-V, while FDS programs are written in FORTRAN without the use of embedded list processing features. This is solely a technological advance, as undoubtedly the GPS could be duplicated in this way. Having a machine efficient program has permitted us to study program performance over a much wider range of problems than would have been the case had we used an interpretive language.

The more interesting difference is in how the programs define differences between states and use them to select operators. The GPS user must provide a table called the "Operator-difference table", and a set of subroutines representing operators and differences. These define the differences which the program can notice and the rewriting rules to be considered in reducing each difference. The FDS, on the other hand, generates its own equivalent of differences and operator-difference tables by analysis of rewriting rules. A FDS program will not generate all the "differences" which a human might see between two states, as we showed in section 8. On the other hand, FDS provides a completely algorithmic theorem proving procedure once the axiomatic system has been defined to the program, while in GPS there is one more step at which human ingenuity or lack of it, can intervene, (6). The question "Which theorem prover is better?" is an unanswerable one, since the two procedures make a different division of labor between man and computer on the theorem proving task.

The later versions of GPS (1,2) have introduced a new sort of goal, "Select the element of the set S which best fulfills criterion C ." This method of defining a goal is quite different from the definitions used in FDS, as "best" implies an ordering of objects on the basis of their relative possession of some property. There is no way of representing such a goal in the FDS structure. It appears from the examples given in (2) that this goal is quite useful, on some problems, but that these problems are somewhat outside the field of theorem proving.

10.2 <u>The resolution principle</u>: The Resolution Principle (10) is a quite different, and very powerful, approach to theorem proving. There are three steps in using this principle. First, one must represent the relevant area of mathematics as a set of clauses in the first order predicate calculus in disjunctive normal form, i.e. as a set of statements of the form (A or B or Not(C)) . In addition to representing the premises of a theorem this way, the negation of the conclusion is also so represented. Using the single rule of inference that if (A or not (B)) and (B or C) are true, then the clause (A or C) may be inferred, a resolution principle program attempts to prove that the negation of the conclusion is inconsistent with the premises.

This technique is very well suited to the discovery of proofs which do not have the line-by-line property. We have already shown that this is precisely the point at which the FDS does poorly. We suspect that there are situations in which the converse is true. One of the problems with the resolution principle method of theorem proving is that the number of inferred clauses multiplies rapidly. A variety of strategies have been suggested for correcting this problem, and some have had notable success (13, 15). Still, we suspect that it will remain a problem. This is particularly so if the basic axioms of the system in which one is working are such that they are clumsy to state in the first order predicate calculus. We offer the Sanderson algebra as an example of such a system. Based on these considerations, we offer the following conjecture, in the hopes that evidence will soon be published.

"If proofs in an area of mathematics typically depend upon inferences from a large number of previously proven theorems or axioms, and if these proofs are likely to display the line-by-line property, the state-to-goal method of theorem proving, as exhibited by FDS or GPS, will be more practical. To the extent that proofs depend upon the convergence of several lines of reasoning, and can be obtained by reference to a relatively small number of previously stated results, the resolution principle will be the more practical technique of theorem proving."

Another contrast between the FDS algorithm and the resolution principle raises quite another question. The resolution principle tends to produce proofs which are certainly valid, (i.e., their validity is provable), but which are hard or impossible for a person to comprehend. Line-by-line proofs are easy to follow. Is this a real question? Robinson (10) correctly states that a theorem is proven if it is algorithmically decideable that its proof is correct, regardless of who comprehends the proof. In some cases, however, the point of obtaining the proof is not to make the assertion but to understand the reasoning which leads to it. If this is the case, the proof must be intelligible.

10.3 Summary: A procedure for mechanical problem solving has been presented. Computer programs based upon this method have proven a large number of simple theorems, and appear to be able to operate at the level of a university undergraduate mathematics student. Illustrations of solutions were given, and the method contrasted to that of the General Problem Solver and the Resolution Principle.

REFERENCES

1. Ernst, G. and Newell, A. Some issues of representation in a General Problem Solver. Proc. Spring Joint Computer Conf., 1967, AFIPS 30.

2. Ernst, G. and Newell, A. Generality and GPS. Carnegie-Mellon University, Department of Computer Science Technical Report, 1967.

3. Greenblatt, R., Eastlake, D. and Crocker, S. The Greenblatt chess program. Proc. Fall Joint Computer Conf. 1967, AFIPS 31, 801-810.

4. Ledley, R. Programming and utilization of digital computers. New York; McGraw-Hill, 1962.

5. Minsky, M. Steps toward artificial intelligence. In Feigenbaum, E. and Feldman, J. (Editors), Computers and Thought, New York; McGraw-Hill, 1963.

6. Newell, A. and Ernst, G. The search for generality. Proc. IFIPS Congress. 1965, 17-24.

7. Newell, A., Shaw, J.C., and Simon, H. Report on a general problem solving program for a computer. Proc. International Conference on Information Processing. Paris; UNESCO House, 1959.

8. Newell, A. and Simon, H.A. GPS, A program that simulates human thought. In Feigenbaum, E. and Feldman, J. (Editors), Computers and Thought, New York; McGraw-Hill, 1963.

9. Quinlan, J.R. A FORTRAN IV general purpose deductive program. Working paper 106, Western Management Science Institute, University of California, Los Angeles, 1966.

10. Robinson, J.A. A machine oriented logic based on the resolution principle. J. ACM. 1965, 12, 23-41.

11. Samuel, A.L. Some studies in machine learning using the game of checkers. In Feigenbaum, E. and Feldman, J. (Editors) Computers and Thought, New York; McGraw-Hill, 1963.

12. Sanderson, J. Theory of programming languages. Ph.D. Thesis, University of Adelaide, 1966.

13. Slagle, J. Automatic theorem proving with renumable and semantic resolution. J. ACM, 1967, 14, 687-697.

14. Suppes, P. and Hill, S. Introductory Logic for the Schools. New York; Random House, 1962.

REFERENCES

15. Wos, L., Carson, D. and Robinson, G. The unit preference strategy in theorem proving. Proc. Fall Joint Computer Conf., 1965, AFIPS 26, 615-621.

16. Wos, L., Carson, D., Robinson, G., and Shalla, L., The concept of demodulation in theorem proving. J. ACM. 1967, 14, [illegible].

FOOTNOTES

1. This research was partially supported by the National Science Foundation, Grant No. NSF B7-1438R, to the University of Washington, Earl Hunt, Principal Investigator, and partially by the Air Force Office of Scientific Research, Office of Aerospace Research, United States Air Force, under AFOSR Grant No. AF-AFOSR-1311-67. Distribution of this document is unlimited.

2. The exact correspondence is difficult to establish, since insofar as we know, no rigorous definition of the algorithms of the General Problem Solver have ever been published. The earlier informal descriptions of GPS (7,8) indicate that the program may have used the recursive method of problem generation defined in section 7.[illegible]. A later paper (11) and technical report (12) suggest that the final version of GPS used something much like the improved, non-recursive procedure of section 7.5.

TABLE 1
MANIPULATION OF + AND -

REWRITING RULES

1	A+B := B+A
2	A+(B+C) := (A+B)+C
3	(A+B)-B := A
4	A := (A+B)-B
5	(A-B)+C := (A+C)-B
6	(A+B)-C := (A-C)+B

THEOREMS

	THEOREM	SECONDS
1	(A+B)+C := A+(B+C)	5
2	(A-B)+B := A	0
3	A := (A-B)+B	0
4	A+(B-C) := (A+B)-C	1
5	(A-B)+C := A+(C-B)	5
6	(A-C)-(B-C) := A-B	215
7	(A+C)-(B+C) := A-B	169
8	A+(B-C) := (A-C)+B	7
9	(A+B)-C := A+(B-C)	4
10	(A-B)-C := (A-C)-B	8
11	A-(B+C) := (A-B)-C	8
12	A+(B-C) := A-(C-B)	6
13	A-(B-C) := A+(C-B)	7
14	(A-B)+C := A-(B-C)	5
15	A-(B-C) := (A-B)+C	4
16	A-(B-C) := (A+C)-B	6
17	(A-B)-C := A-(B+C)	7
18	(A+B)-C := A-(C-B)	1

```
                         TABLE 2
         MANIPULATION OF UNARY MINUS ("NEG") AND 0

                      REWRITING RULES

      1 A+B := B+A                     15 (A+B)-C := A+(B-C)
      2 A+(B+C) := (A+B)+C             16 (A-B)-C := (A-C)-B
      3 (A+B)-B := A                   17 A-(B+C) := (A-B)-C
      4 A := (A+B)-B                   18 A+(B-C) := A-(C-B)
      5 (A-B)+C := (A+C)-B             19 A-(B-C) := A+(C-B)
      6 (A+B)-C := (A-C)+B             20 (A-B)+C := A-(B-C)
      7 (A+B)+C := A+(B+C)             21 A-(B-C) := (A-B)+C
      8 (A-B)+B := A                   22 A-(B-C) := (A+C)-B
      9 A := (A-B)+B                   23 (A-B)-C := A-(B+C)
     10 A+(B-C) := (A+B)-C             24 (A+B)-C := A-(C-B)
     11 (A-B)+C := A+(C-B)             25 A+0 := A
     12 (A-C)-(B-C) := A-B             26 A-0 := A
     13 (A+C)-(B+C) := A-B             27 NEG(A) := 0-A
     14 A+(B-C) := (A-C)+B             28 0-A := NEG(A)

                         THEOREMS

     THEOREM                 SECONDS        THEOREM                   SECONDS
  1 A := A+0                       5    7 A+NEG(B) := A-B                   5
  2 A := A-0                       8    8 A-NEG(B) := A+B                   6
  3 A-A := 0                      21    9 NEG(A)+NEG(B) :=NEG(A+B)        125
  4 0 := A-A                       6   10 NEG(A)-NEG(B) :=NEG(A-B)         47
  5 A+NEG(A) := 0                  5   11 NEG(NEG(A)) := A                105
  6 0 := A+NEG(A)                 58
```

TABLE 3
SANDERSON ALGEBRA

REWRITING RULES

1 (A+B)+C := A+(B+C)
2 A+(B+C) := (A+B)+C
3 I+A := A
4 A := I+A
5 A+I := A
6 A := A+I
7 Z+A := Z
8 A+Z := Z
9 Z := A+Z
10 Z := Z+A
11 I/I := I
12 I := I/I
13 (A+C)/(B+C) := (A/B)+C
14 (A/B)+C := (A+C)/(B+C)
15 (A/B)/C := A/C
16 A/C := (A/B)/C
17 A/(B/C) := A/C
18 A/C := A/(B/C)
19 SIG(A) := (A+SIG(A))/I
20 (A+SIG(A))/I := SIG(A)
21 SIG(A/B) := SIG(A)
22 SIG(A) := SIG(A/B)
23 SIG(A)+B/C := SIG(A)+C
24 SIG(A)+C := SIG(A)+B/C
25 SIG(I) := Z/I
26 Z/I := SIG(I)

THEOREMS

	THEOREM	SECONDS
1	A/A := A	1
2	A := A/A	2
3	((A/B)+C)/D := (A+C)/D	2
4	A/((B/C)+D) := A/(C+D)	2
5	(A+B)/C := ((A/D)+B)/C	10
6	A/(B+C) := A/((D/B)+C)	21
7	SIG(Z) := Z/I	1
8	Z/I := SIG(Z)	1
9	SIG(A)/I := SIG(A)	13
10	SIG(A) := SIG(A)/I	11
11	SIG(A)+SIG(B) := SIG(A)	36
12	SIG(A) := SIG(A)+SIG(B)	596
13	SIG(A)/SIG(B) := SIG(A)	2
14	SIG(A) := SIG(A)/SIG(B)	1334
15	SIG(SIG(A)) := SIG(A)	108

TABLE 4

PROPOSITIONAL CALCULUS

REWRITING RULES

1.	P·Q	:=	Q·P	4.	–Q·(P->Q)	:=	–P
2.	P·(Q·R)	:=	(P·Q)·P	5.	– –P	:=	P
3.	P·(P->Q)	:=	Q	6.	P	:=	– –P

THEOREMS

THEOREM				SEC
1.	(S->–R)·R	:=	–S	1
2.	B·(–A->–B)	:=	A	1
3.	–B·(A->B)·(–A->C)	:=	C	2
4.	–C·(B->C)·(–B->–A)	:=	A	2
5.	(P->–Q)·Q·(–P->(R·S))	:=	R·S	6

TABLE 5

INEQUALITIES

REWRITING RULES

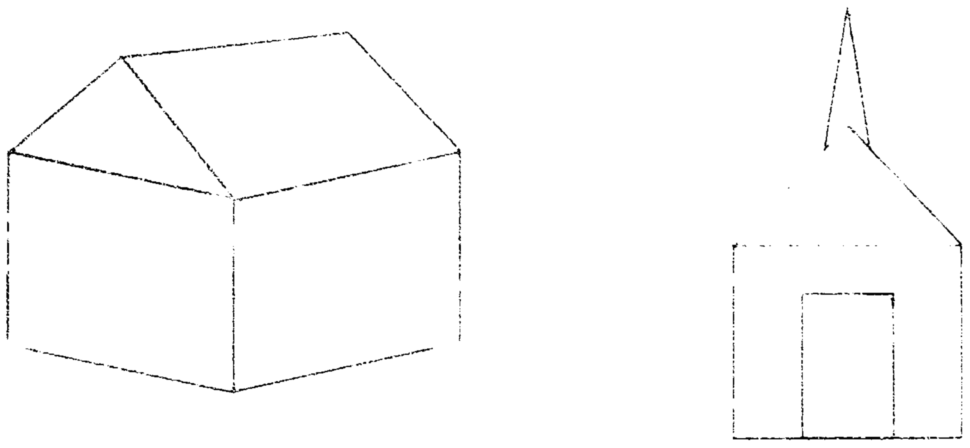
1. A > B := (A-B) > 0
2. (A-B) > 0 := A > B
3. A > 0 := 0 > NEG(A)
4. 0 > NEG(A) := A > 0
5. A IS REAL := (A>0) | (A=0) | (NEG(A) > 0)
6. (A IS REAL)·(B IS REAL) := (A-B) IS REAL
7. (A IS REAL)·(B IS REAL) := (A+B) IS REAL
8. (A-B)=0 := A=B
9. A=B := (A-B)=0
10. (A>0)·(B>0) := (A+B)>0
11. (A>0)·(B>0) := (AxB)>0

IN ADDITION, 123 AXIOMS OF LOGIC AND ARITHMETIC WERE GIVEN.

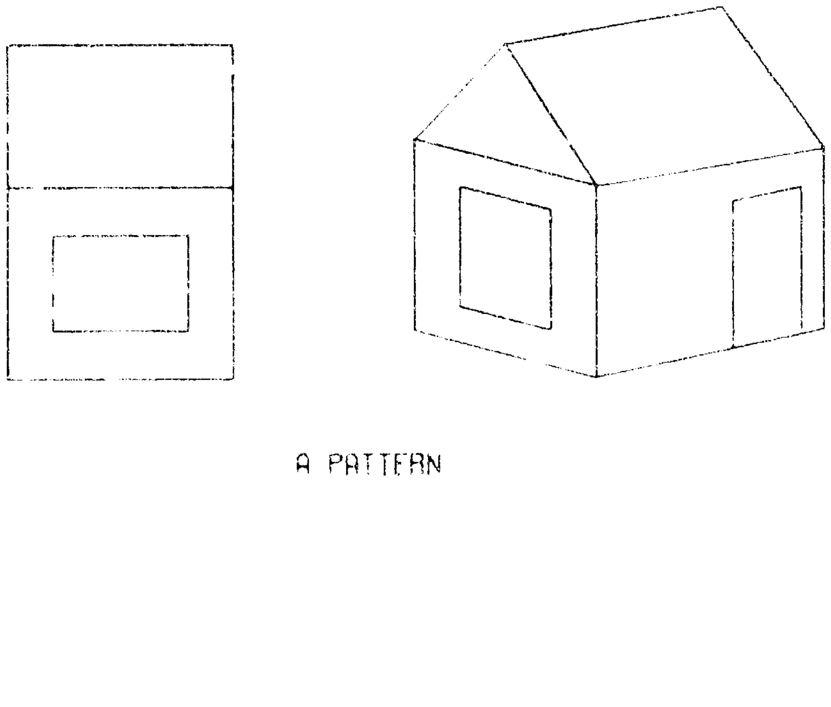
THEOREMS

THEOREM	SECS
1. 0>A := NEG(A)>0	10
2. NEG(A)>0 := 0>A	10
3. 0>(A-B) := B>A	55
4. A>B := 0>(B-A)	163
5. (A-B)>0 := 0>(B-A)	32
6. 0>(B-A) := (A-B)>0	50
7. NEG(A-B)>0 := B>A	15
8. B>A := NEG (A-B)>0	11
9. (A IS REAL)·(B IS REAL) := ((A-B)>0)|((A-B)=0)|(NEG(A-B)>0)	10
10. (A IS REAL)·(B IS REAL) := (A>B)|(A=B)|(B>A)	143

FIGURE 1



TWO OF THE BASIC PATTERN TYPES



A PATTERN

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) University of Washington Psychology Department Seattle, Washington	2a. REPORT SECURITY CLASSIFICATION Unclassified
	2b. GROUP -----

3. REPORT TITLE

A FORMAL DEDUCTIVE PROBLEM-SOLVING SYSTEM

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific Interim

5. AUTHOR(S) (First name, middle initial, last name)

J. R. Quinlan and E. B. Hunt

6. REPORT DATE February 9, 1968	7a. TOTAL NO. OF PAGES 39	7b. NO. OF REFS 16
8a. CONTRACT OR GRANT NO. AF-AFOSR-1311-67	9a. ORIGINATOR'S REPORT NUMBER(S) 68-1-01	
b. PROJECT NO. ---- 9778-01		
c. 6144501F	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) AFOSR 68-0825	
d. 681313		

10. DISTRIBUTION STATEMENT

1. This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTE -------- TECH, OTHER	12. SPONSORING MILITARY ACTIVITY Air Force Office of Scientific Research Office of Aerospace Research United States Air Force (SRLB)

13. ABSTRACT

A formal description of a generalized theorem proving computer program is given. The program has been written in Fortran IV, but this description is concerned with logical flow and definitions of the algorithm, rather than programming details. Examples of performance of the program are given, using several fields of mathematics for illustrative purposes.

DD FORM NOV 65 1473

Security Classification

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computer Program Theorem proving Mathematics Artificial Intelligence Heuristic Programming						

Security Classification